Algorithms and Data Structures 2
CS 1501

Spring 2022

Sherif Khattab

ksm73@pitt.edu

(Slides are adapted from Dr. Ramirez's and Dr. Farnan's CS1501 slides.)

Announcements

- Upcoming deadlines:
  - Lab 8 due on 3/25
  - Homework 9 due on 3/28
  - Assignment 2 due on 3/28
  - Lab 9 due on 4/1

Previous Lecture …

- Prim's and Kruskal's MST algorithms

CourseMIRROR Reflections (most confusing)

- What is the best edge? The one before the vertex or the one after?
- Prims algorithm was very confusing
- Determining the runtime of prims algorithm was confusing
- The method of calculating a low value and num value for a specific vertex of a graph was most confusing. Additionally, the calculation of the MST of a graph was also confusing.
- I thought the algorithm to construct the Prims was a bit confusing
- Still a missing something with articulation point algorithm. Hope to have a homework problem to work through
- I was confused about which vertex to check edges from in Prims Algorithm

CourseMIRROR Reflections (most confusing)

- For the non-naïve Prim's algorithm, how are best edge and parent array values determined and then later overwritten?
- Why we multiply 2 in the runtime analysis of Best edge searching for graph implemented by matrix prim's algorithm to find mst
- The order of using prims algorithm
- Finding the minimum edge value for each node in the traversal of the new Prim's algorithm

CourseMIRROR Reflections (most interesting)

- prim algorithm run time
- optimizing Prim’s algorithm with the parent and best edge arrays
- That we are able to cut down the runtime by only seeing the best edges
- I found it interesting how Kruskal Algorithm used a priority queue to solve the minimum spanning tree problem
- Tracing through the algorithms
- minimum spanning tree

CourseMIRROR Reflections (most interesting)

- How DFS can find articulation points through the low and num values
- The new articulation point retrieval example was very helpful to go through
- I thought the way you are able to find the lowest cost by such a simple algorithm is interesting
- Kruskals algorithm seems interesting. Want to see more of it next lecture
- The possibilities of prims

Repetitive Minimum Problem

- Input:
  - a (large) dynamic set of data items in the form of
- Output:
  - find a minimum item
- You are implementing an algorithm that repeats this problem
  - examples of such an algorithm?
    - Prim’s, Huffman tree construction
- What we cover today applies to the repetitive maximum problem as well

Let’s create an ADT!

- The Priority Queue ADT

- Primary operations of the PQ:
    - Insert
    - Find item with highest priority
        - e.g., findMin() or findMax()
    - Remove an item with highest priority
        - e.g., removeMin() or removeMax()
- We mentioned priority queues in building Huffman tries
- How do we implement these operations?
    - Simplest approach: arrays

Unsorted array PQ

- Insert:
  - Add new item to the end of the array
  - $\Theta(1)$
- Find:
  - Search for the highest priority item (e.g., min or max)
  - $\Theta(n)$
- Remove:
  - Search for the highest priority item and delete
  - $\Theta(n)$

- Runtime for use in Huffman tree generation?

Sorted array PQ

- Insert:
  - Add new item in appropriate sorted order
  - $\Theta(n)$
- Find:
  - Return the item at the end of the array
  - $\Theta(1)$
- Remove:
  - Return and delete the item at the end of the array
  - $\Theta(1)$

- Runtime for use in Huffman tree generation?

Amortized Runtime

$$\text{Amortized runtime} = \frac{\text{Total runtime of a sequence of operations}}{\#\text{operations}}$$

Amortized Time

n inserts $\qquad\qquad$ n DeleteMin

$\Theta(n)$ $\qquad\qquad$ $\Theta(1)$

$$n^2 + n = \Theta(n^2)$$

$$\text{Amortized Time} = \frac{n^2}{\#\text{operations}}$$

$$= \frac{n^2}{2n} = \Theta(n)$$

So what other options do we have?

- What about a binary search tree?
  - Insert
    - Average case of $\Theta(\lg n)$, but worst case of $\Theta(n)$
  - Find
    - Average case of $\Theta(\lg n)$, but worst case of $\Theta(n)$
  - Remove
    - Average case of $\Theta(\lg n)$, but worst case of $\Theta(n)$
- OK, so in the average case, all operations are $\Theta(\lg n)$
  - No constant time operations
  - Worst case is $\Theta(n)$ for all operations

Is a BST overkill?

- Our find and remove operations only need the highest priority item, not to find/remove *any* item
    - Can we take advantage of this to improve our runtime?
        - Yes!

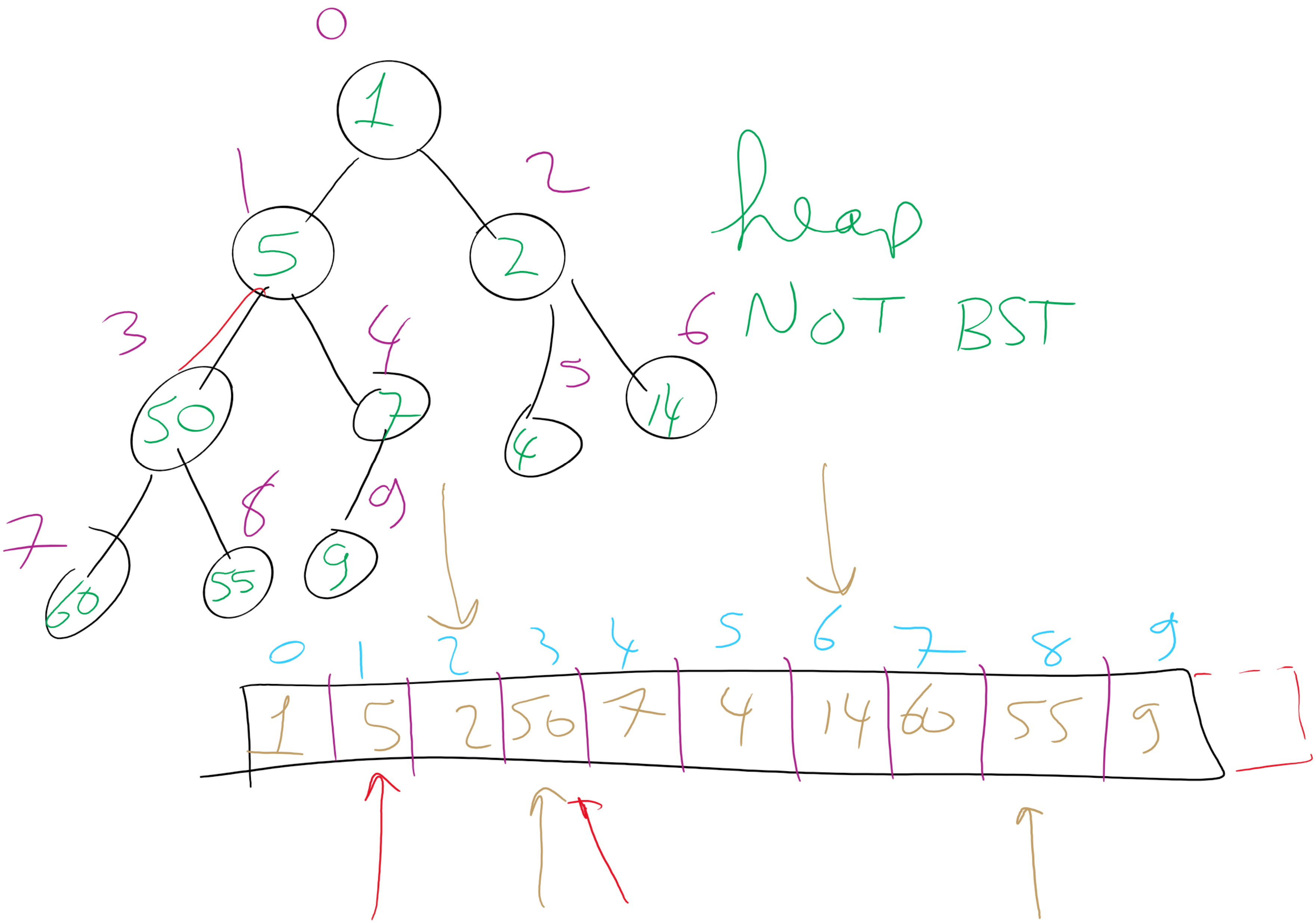
The heap

- A heap is complete binary tree such that for each node T in the tree:
  - T.item is of a higher priority than T.right_child.item
  - T.item is of a higher priority than T.left_child.item

- It does not matter how T.left_child.item relates to T.right_child.item
  - This is a relaxation of the approach needed by a BST

The *heap property*

Heap Example



Heap PQ runtimes

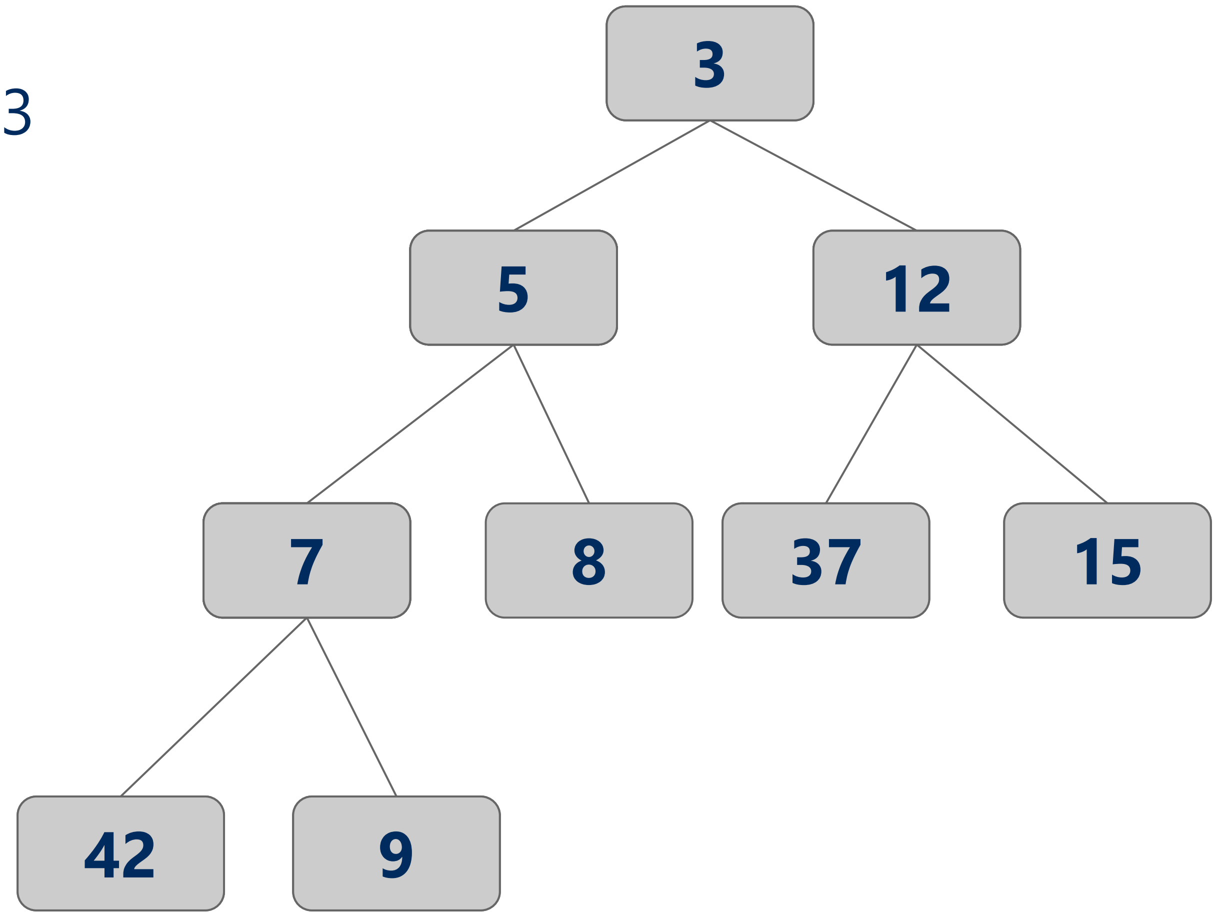
- Find is easy
  - Simply the root of the tree
    - $\Theta(1)$
- Remove and insert are not quite so trivial
  - The tree is modified and the heap property must be maintained

Heap insert

- Add a new node at the next available leaf
- Push the new node up the tree until it is supporting the heap property

Min heap insert

Insert:
7, 42, 37, 5, 8, 15, 12, 9, 3

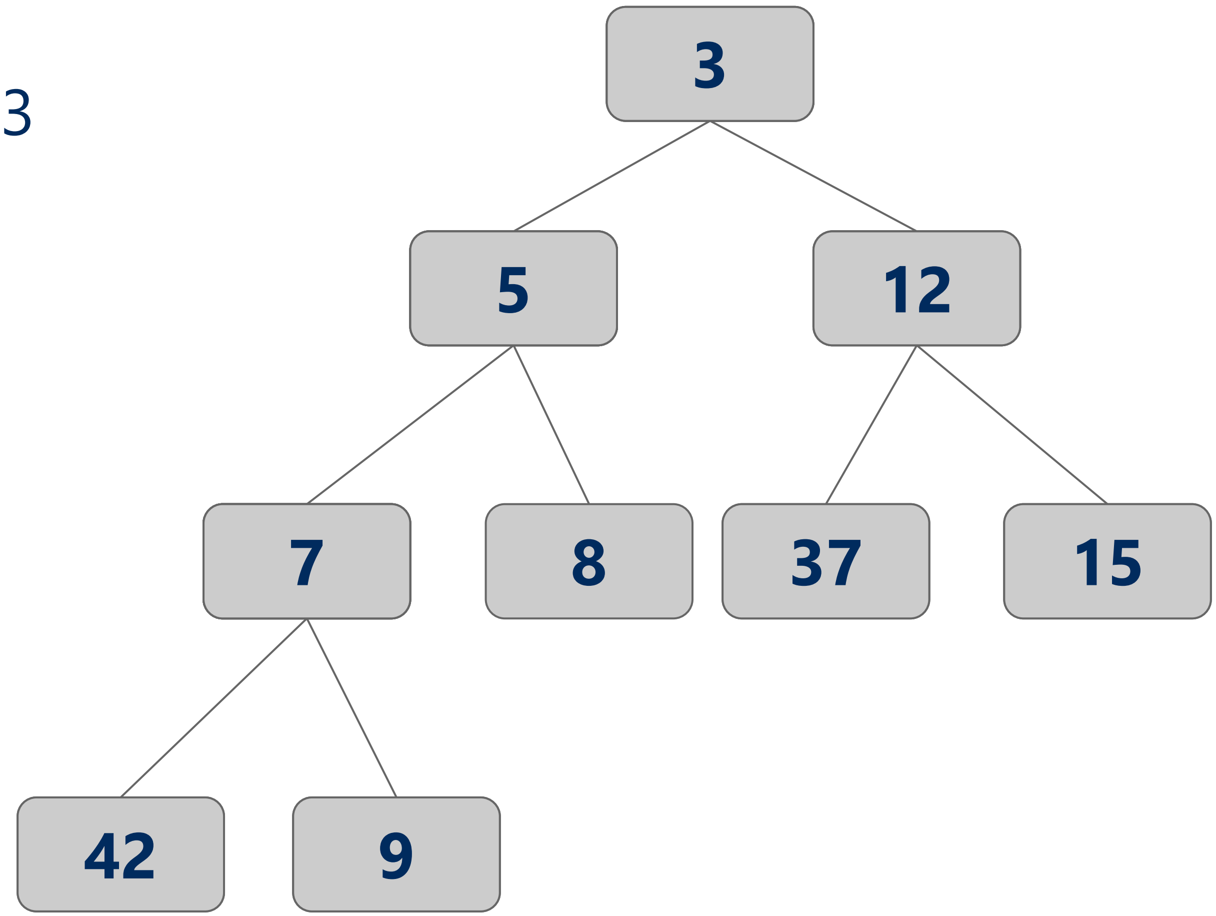


Heap insert

- Add a new node at the next available leaf
- Push the new node up the tree until it is supporting the heap property

Min heap insert

Insert:
7, 42, 37, 5, 8, 15, 12, 9, 3

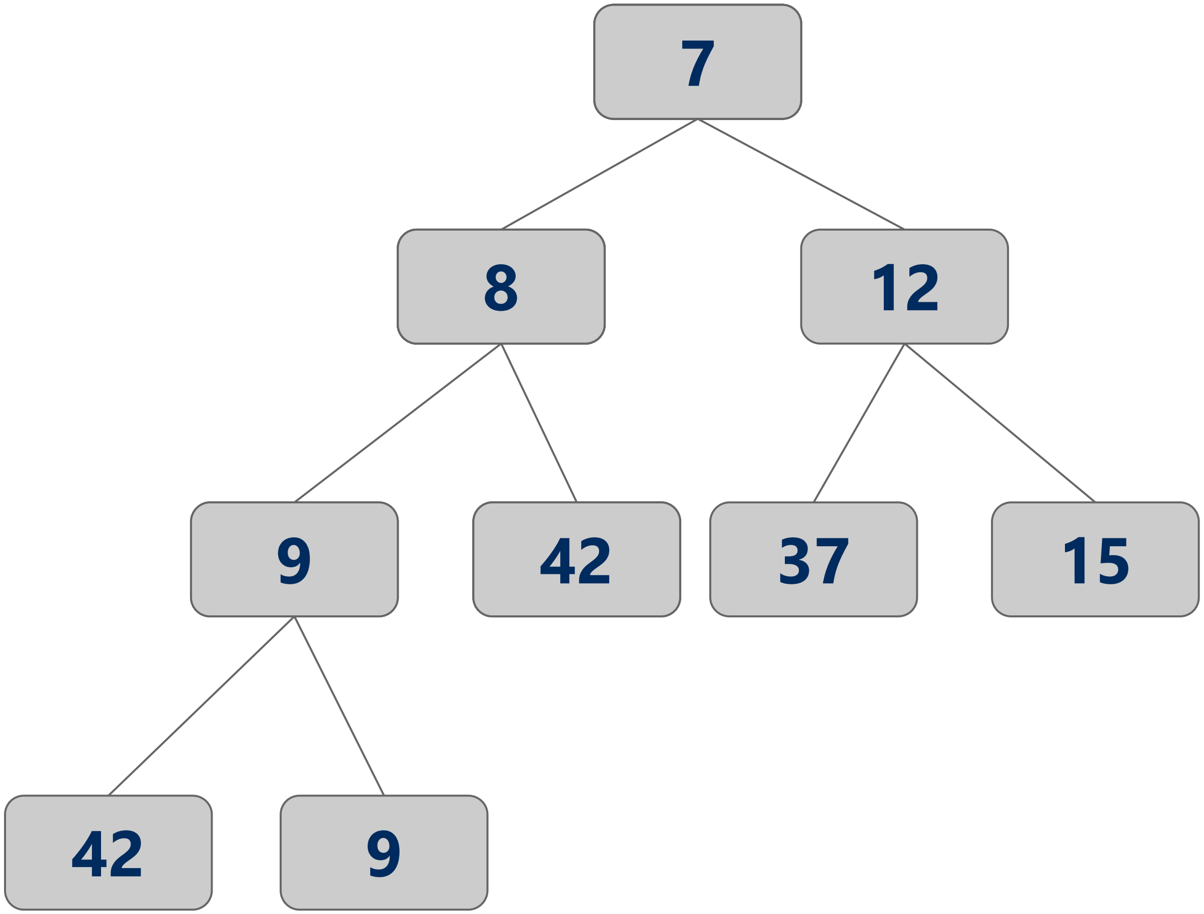


Heap remove

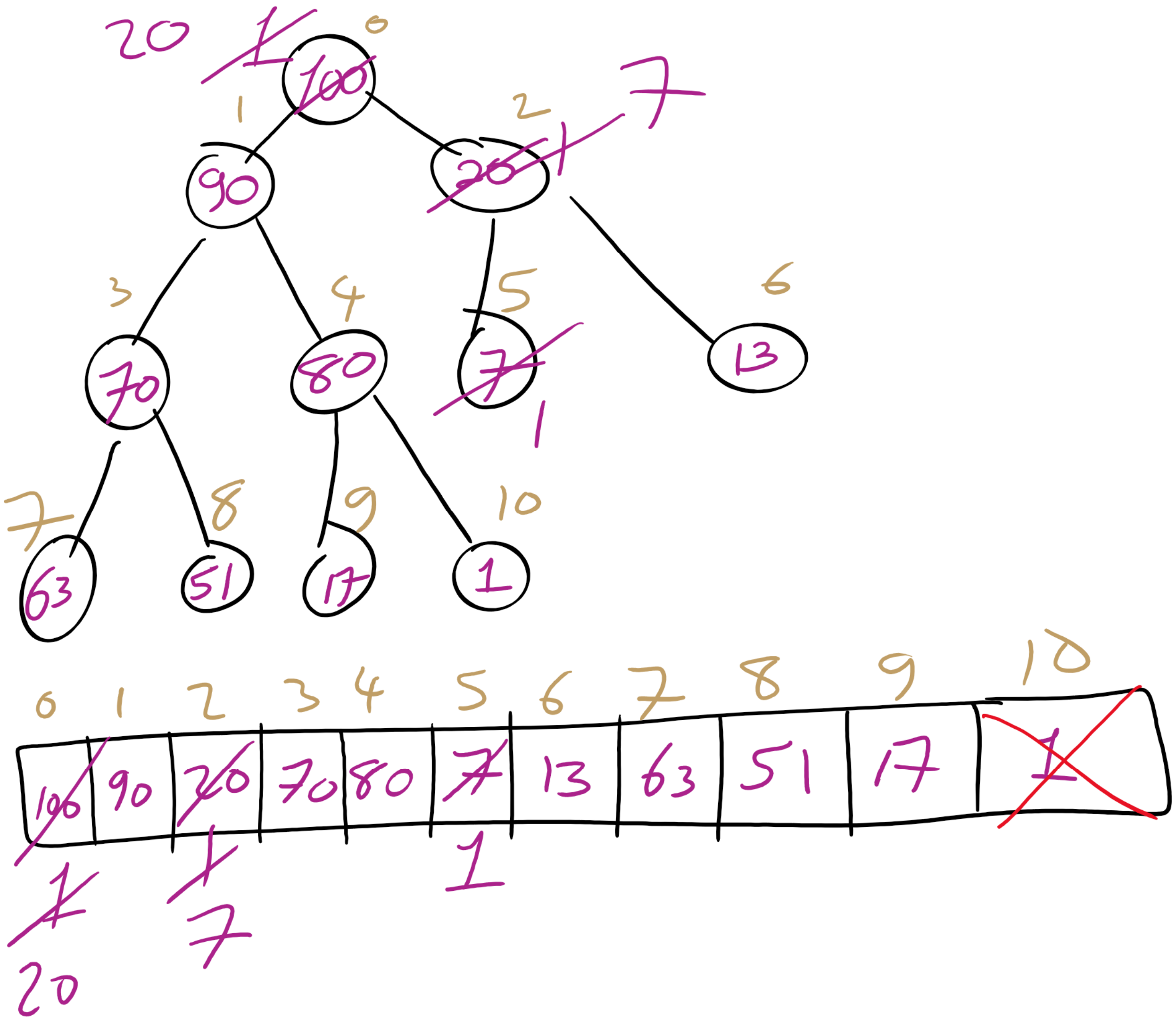
- Tricky to delete root…
  - So let's simply overwrite the root with the item from the last leaf and delete the last leaf
    - But then the root is violating the heap property…
      - So we push the root down the tree until it is supporting the heap property

Min heap removal

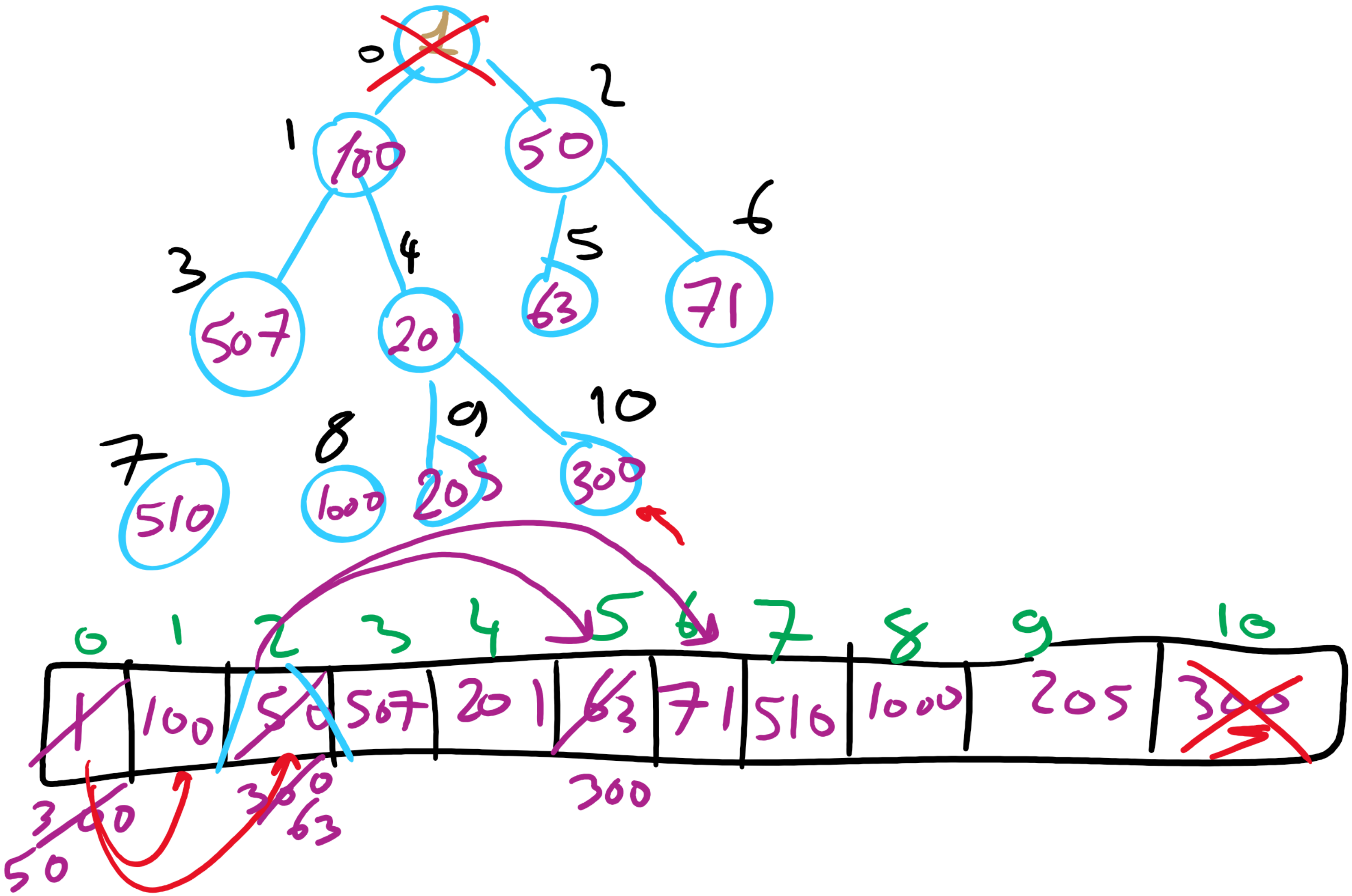
NO!



Heap removeMax Example



Heap removeMin Example



Heap runtimes

- Find
  - $\Theta(1)$
- Insert and remove
  - Height of a complete binary tree is lg n
  - At most, upheap and downheap operations traverse the height of the tree
  - Hence, insert and remove are $\Theta(\lg n)$

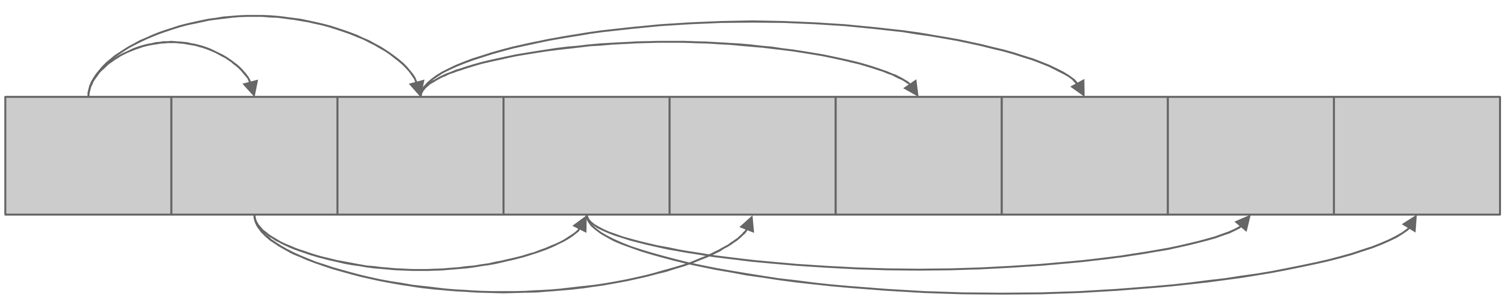
Heap implementation

- Simply implement tree nodes like for BST
    - This requires overhead for dynamic node allocation
    - Also must follow chains of parent/child relations to traverse the tree
- Note that a heap will be a complete binary tree…
    - We can easily represent a complete binary tree using an array

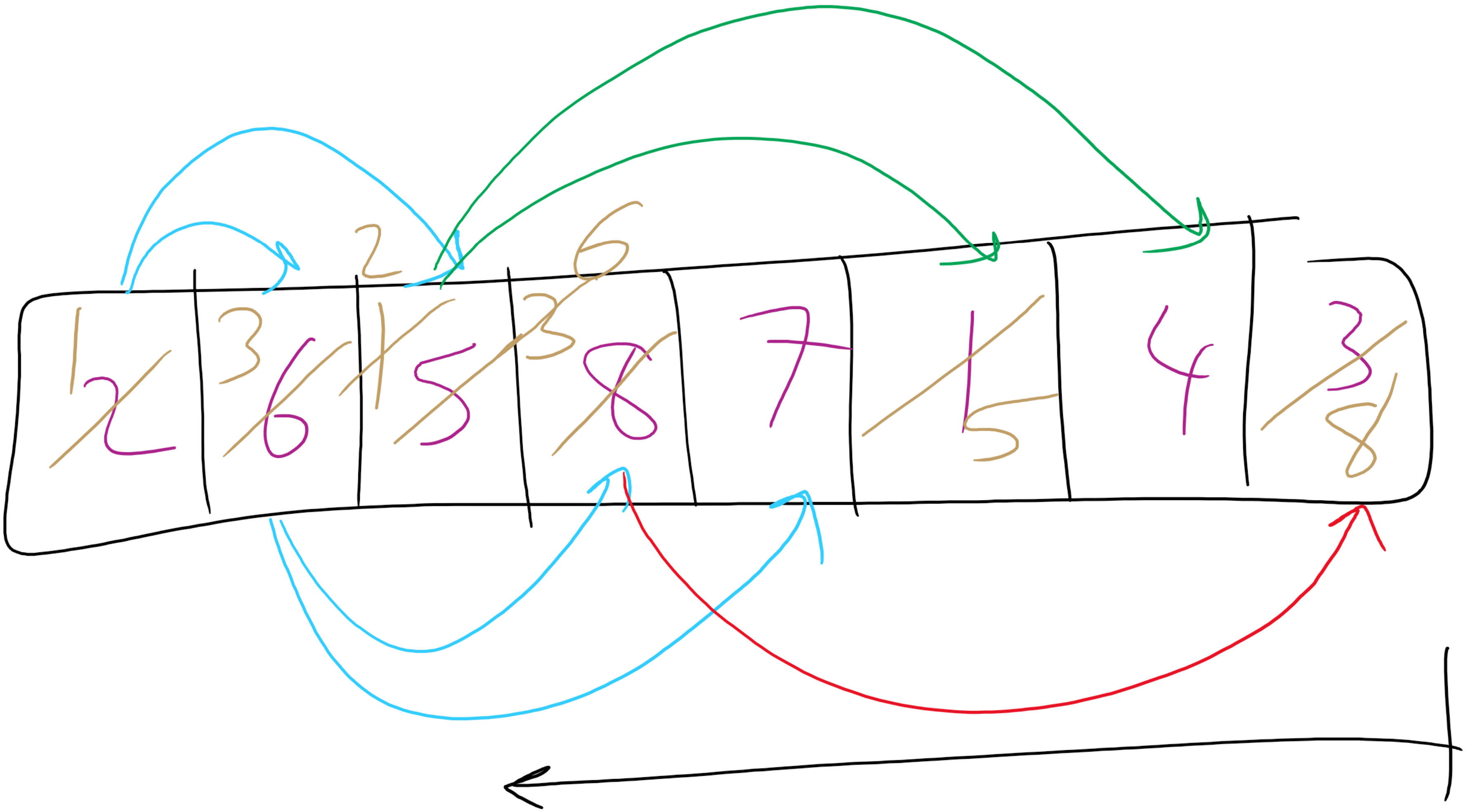
Storing a heap in an array

- Number nodes row-wise starting at 0
- Use these numbers as indices in the array
- Now, for node at index i
  - $\text{parent}(i) = \lfloor (i - 1) / 2 \rfloor$
  - $\text{left\_child}(i) = 2i + 1$
  - $\text{right\_child}(i) = 2i + 2$

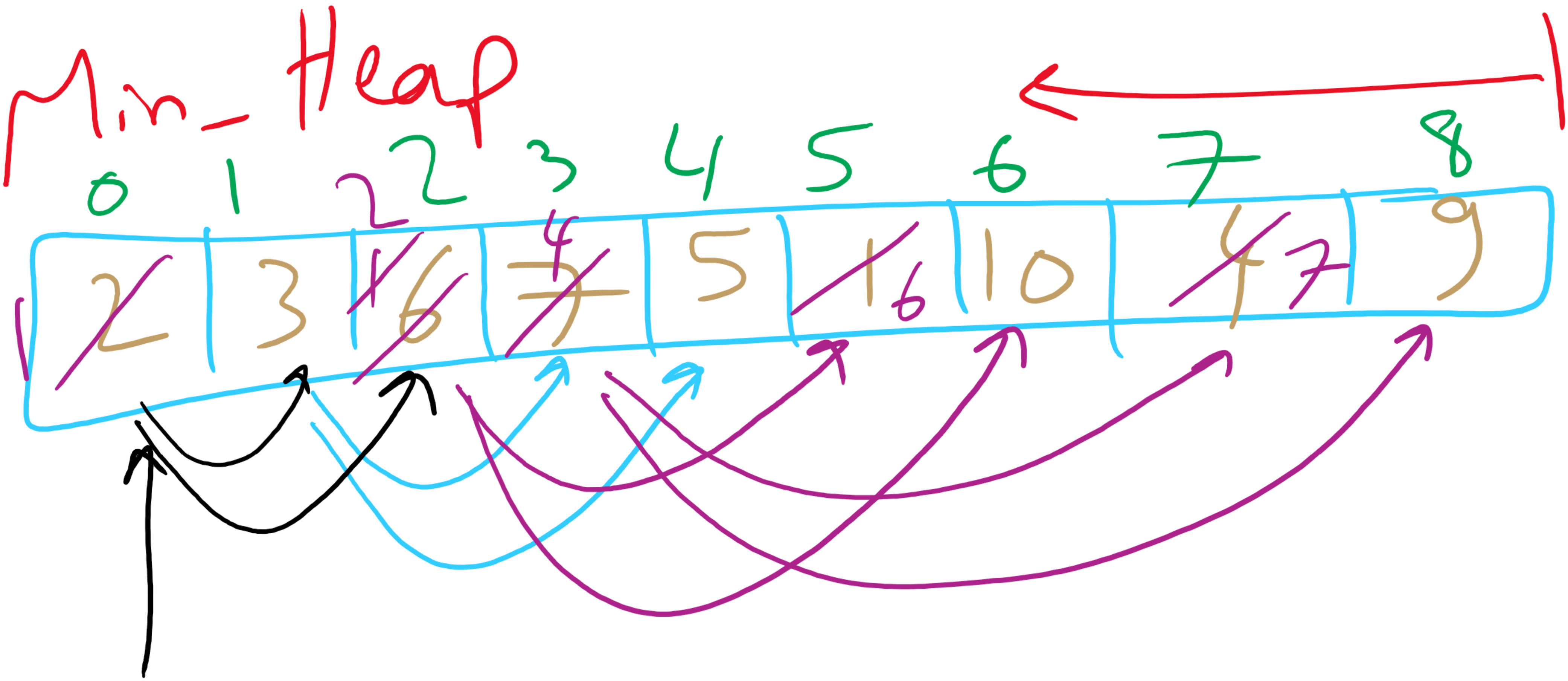
For arrays indexed from 0



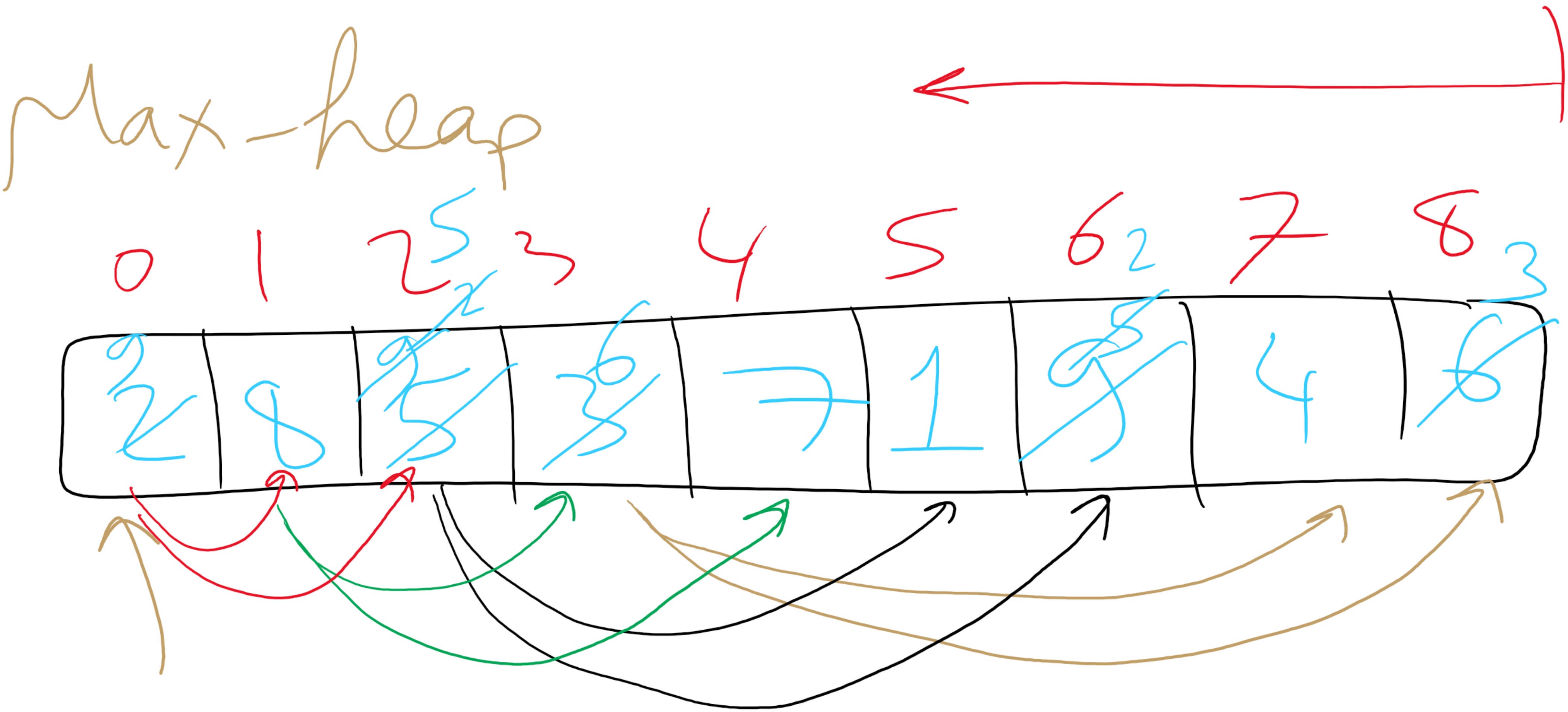
Heapify Operation



Heapify Example



Heapify Example



HeapSort Pseudo-code

HeapSort(Array a)

- Heapify (a) —— $\Theta(n)$

ascending MAX_HEAP

descending MIN_HEAP

end = n - 1

for i = 0 ... n - 1

- remove highest priority item by swapping with the end of the array —— log n
- end--

n

Heap Sort

- Heapify the numbers
  - MAX heap to sort ascending
  - MIN heap to sort descending
- "Remove" the root
  - Don’t actually delete the leaf node
- Consider the heap to be from 0 .. length - 1
- Repeat

Heap sort analysis

- Runtime:
  - Worst case:
    - n log n
- In-place?
  - Yes
- Stable?
  - No

Storing Objects in PQ

- What if we want to update an Object?
  - What is the runtime to find an arbitrary item in a heap?
    - $\Theta(n)$
    - Hence, updating an item in the heap is $\Theta(n)$
  - Can we improve of this?
    - Back the PQ with something other than a heap?
    - Develop a clever workaround?

Please submit your reflections by using the CourseMIRROR App

If you are having a problem with CourseMIRROR, please send an email to coursemirror.development@gmail.com